1. Introducción

El paradigma de la programación orientada a objetos, también conocida como POO, es la que usa objetos y sus interacciones para diseñar aplicaciones y programas de computadora [1]. Desde su nacimiento en los años 60´s, el uso de la POO se ha incrementado notablemente, esto se ha debido principalmente al nacimiento de distintos lenguajes de programación orientado a objetos que ganaron bastante popularidad como C++, Java y más recientemente el .NET Framework, incluyendo sus lenguajes C# y Visual Basic .NET [2]. Por lo tanto, la POO se ha convertido en el estándar de facto en la industria de software, sin embargo, ni Java ni los lenguajes oficiales del .NET Framework son a prueba de balas. Existen muchos programadores sin el conocimiento y la experiencia necesaria que están creando programas mal diseñados, los cuales se traducen a aplicaciones ineficientes y difíciles de mantener y extender. Incluso, desarrolladores con la experiencia y habilidades necesarias parar desarrollar aplicaciones extensibles mediante la aplicación de

distintas técnicas conocidas, se han encontrado en el dilema de que sus clientes necesitan extender sus componentes en formas inesperadas al diseño inicial.

Aunque ambas situaciones parecieran distintas en un inicio, una se debe a la falta de conocimiento y experiencia en el área, mientras que la otra es por falta de planeación ó cambio de planes; ambas se pueden mejorar mediante la reducción de las dependencias excesivas entre sus componentes y lo que por consecuencia minimizará el acoplamiento entre ellos, y resultará finalmente en una alta cohesión.

Existen varias razones importantes por la cual deberíamos evitar tener demasiadas dependencias cuando desarrollamos un sistema orientado a objetos. La primera es porque las dependencias previenen la extensibilidad. Tener muchas dependencias rompe el viejo principio que dice: “Las entidades de software deberían estar abiertas para la extensión, pero cerradas para su modificación” [15]. Una razón por la que se tienen una gran cantidad de dependencias es el uso impropio de la abstracción de objetos, y las áreas donde no se usan estas abstracciones son precisamente las más difíciles de extender. Además de esto, las dependencias inhiben también la reusabilidad. Para lograr altos niveles de reusabilidad se debe tener mucho cuidado en la estructura del componente y sobre todo a la unidad de despliegue. La mayoría del software con una estructura compleja y muchas dependencias físicas minimizan la probabilidad de lograr un grado alto de reusabilidad. Otra razón, es que las dependencias restringen la facilidad de aplicar pruebas automatizadas, esto se debe principalmente al gran acoplamiento entre sus componentes y clases, lo que impide la habilidad de probar una clase en específico independientemente de las demás. Finalmente, las dependencias también limitan el entendimiento general de la aplicación. Cuando se trabaja en el desarrollo de software, es muy importante entender la arquitectura estructural

del sistema, así como las construcciones del diseño. Una estructura con dependencias complejas es normalmente más difícil de entender.

Afortunadamente, existe una solución para estos problemas, la cual es conocida como la refactorización. Refactorizar es el proceso de cambiar un software de tal forma que el comportamiento externo no cambia, más bien, se mejora su estructura interna [3]. En otras palabras, cuando un desarrollador refactoriza su aplicación, lo que realmente hace es mejorar el diseño del código fuente una vez que ha sido escrito. Al principio de la industria del software, solo desarrolladores expertos habían usado técnicas para mejorar la integridad estructural de sus aplicaciones, ahora, existe una extensa documentación [4] y libros [3] que describen estas técnicas a fondo. El propósito de estos documentos es el de demostrar a los desarrolladores de software como pueden tomar un mal diseño de software y convertirlo en un buen diseño, con código robusto.

Desgraciadamente, aunque la refactorización aparentemente resuelve completamente el problema, la mayoría de las técnicas descritas por la literatura solo se pueden aplicar cuando se tiene acceso directo al código fuente de la aplicación, ignorando las veces cuando los desarrolladores ocupan librerías externas, tienen solo una versión ejecutable del componente, o simplemente el código fuente ya no es comprensible. En estos casos, las técnicas de refactorización se vuelven inútiles.

Con esto en mente, estamos proponiendo una nueva herramienta bajo el nombre de Afterex, que tiene el propósito de romper dependencias entre ensamblados .NET mediante la refactorización de su código intermedio (CIL), dándole la libertad a los desarrolladores de poder intercambiar implementaciones de tipos concretas en el mejor caso, o por lo

menos proveer de una capa más abstracta. Nuestra investigación está concentrada específicamente en dos tipos de dependencias. La primera de ellas, a la que nos referiremos como dependencia contenida, es en la que una dependencia entre 2 archivos DLL es únicamente debido al hecho de que una de ellas tiene un método o más, dónde un tipo del segundo es instanciado. El segundo caso es el conocido dependencia directa, ésta se da cuando la dependencia es expuesta directamente por clase, como valor de retorno en un método, como parámetro de un método e incluso como clase base.

El objetivo primordial de esta tesis es el de explorar y estudiar los problemas y asunciones realizadas al momento de crear una herramienta que trate romper dependencias de este tipo. Desafortunadamente, una herramienta de este tipo está atada con el lenguaje de programación o plataforma, sin embargo, las ideas básicas pueden ser fácilmente transferibles a otras tecnologías basadas en la POO. Por esta misma razón, escogimos explorar estas técnicas usando el .NET Framework creado por Microsoft. El .NET Framework provee un lenguaje intermedio, el cuál es el lenguaje base para el CLI (Common Language Infrastructure), y dado que es ejecutado por una máquina virtual, diferentes lenguajes de programación puede ser desarrollados para generar CIL y ser ocupados por el .NET Framework, por lo cual los ensamblados .NET son ideales para explorar este tipo de situaciones.

1.1 Objetivos.

1.1.1 Objetivo general.

Estudiar la refactorización de ensamblados binarios con el objetivo de mejorar la reusabilidad y capacidad de resolución de frameworks. Para lograr esto, se desarrollará una

herramienta que refactoriza ensamblados existentes, con el objetivo de romper las dependencias directas y contenidas entre distintas clases, y hacer posible satisfacer estas dependencias con otros tipos, mediante la mejora de la modularización y extensibilidad de los componentes.

La refactorización es una seria de transformaciones que corrigen y preservan ciertas propiedades, pero el resultado de salida es una aplicación con un código más general que el original, que en nuestro caso específico es un código sin una dependencia en específico. Por tanto no podemos solo afirmar que la transformación realizada por una refactorización T en un programa P tiene las mismas propiedades R antes y después de la refactorización, sino que las propiedades R' en el programa refactorizado P' deben ser por lo menos equivalentes a R. Formalmente podríamos definir el objetivo de la siguiente forma:

Consideremos una aplicación como una tupla (P, D) en la cual P es la versión binaria del programa y D es el conjunto de dependencias de éste. Dada un una aplicación (P, D) y una dependencia d, tal que $D \supseteq d$ y $P \rightarrow R$ donde R son propiedades del programa, proporcionaremos una función de refactorización $T((P, D), d) = [\![P]\!]', D')$, donde $P' \rightarrow R', \quad R' \equiv R \; y \; d \notin D'$.

Esto quiere decir que proporcionaremos una función que recibe tres parámetros, el programa encapsulado en un DLL, el cual incluye las dependencias, y la dependencia que deseamos romper. La salida será un programa que tiene las mismas propiedades que el original por lo menos, pero ahora la dependencia indicada ya no se encuentra en el nuevo programa. Esto da cabida a que el nuevo programa pueda cambiar esta dependencia, sin embargo el programa de salida, debe funcionar como en el programa original.

1.1.2 Objetivos específicos.

- Una solución completa para el escenario de dependencias contenidas.
- Analizar el escenario de dependencias directas.
- Un API reusable, siguiendo las mejores prácticas dictadas por el Framework Design Guidelines [17].
- Aplicaciones ejemplo que demuestren las características más importantes de la herramienta así como la utilidad del mismo.
- Un análisis detallado de las asunciones realizadas así como de los problemas encontrados.
- Una herramienta extensible que permite a los desarrolladores terceros mejorarla mediante sus propias reglas de refactorización mediante una arquitectura basada en extensiones.

1.2 Alcances y limitaciones.

Esta tesis está concentrada en trabajar con ensamblados .NET y el .NET Framework. Esto significa que la herramienta soportará cualquier lenguaje de programación que esté soportado por el .NET Framework. Sin embargo, trasladar la idea a otro entorno administrado ó lenguaje de programación orientado a objetos, debería ser fácil con las herramientas indicadas, principalmente porque los algoritmos y patrones utilizados, no son específicos a ningún lenguaje de programación o tecnología.

Hay varias diferentes formas de crear dependencias; algunas de ellas más benignas que otras. Para el objetivo de esta tesis solo trabajaremos con las dependencias

denominadas contenidas y directas. Hay principalmente dos razones para esto, la primera es porque estas son las más comunes en la industria del software, y segundo porque estas son las más abordables. La dependencia de tipo contenido es la que una clase tiene, pero que no es comunicada externamente. La clase usa el objeto contenido, y por tanto es dependiente a él, sin embargo, la dependencia no es propagada a los usuarios de la clase original. Por otro lado, la dependencia de tipo directa es cuando es expuesta por una clase, ya sea como valor de retorno, parámetro ó clase base. Existen otros tipos de dependencias como la dependencia indirecta que sucede cuando una dependencia es expuesta por otra dependencia, también se encuentran las dependencias escondidas que son un caso especial de las dependencias contenidas pero estas pueden causar que un componente falle. Ciertamente, trabajar con las últimas dos excede el tiempo disponible para acabar esta tesis, ya que los factores con los que se tienen que trabajar son muchos más grandes, haciendo muy difícil su estudio.

La investigación esta divida por cada uno de los escenarios, aunque en realidad estos pueden vivir juntos en una misma aplicación. Para el caso de las dependencias contenidas planeamos dar una solución completa al usuario. Por otro lado, para el caso de dependencias directas, dado que está compuesto por diferentes sub-casos, como por ejemplo parámetros de retorno, excepciones, tipos genéricos, clases base, etc. Debido a las restricciones de tiempo, solo fue posible abordar el problema, dejando las bases para futuras investigaciones.

También se crearan dos ejemplos de aplicaciones reales para el caso de dependencias basadas en implementación, esto con el propósito de demostrar las características más importantes de la herramienta en escenarios reales.

Finalmente, dado que nuestro objetivo principal son estudiar y desarrollar una aplicación para romper dependencias, no hubo necesidad de crear una interfaz gráfica para dicho programa, a cambio, estamos proponiendo una aplicación basada en consola, que es extensible mediante una arquitectura de plug-ins, y un API con métodos que encapsulan la modificación y creación de lenguaje intermedio, dándole la posibilidad a terceros de añadir sus propias reglas de refactorización, y poder continuar con la investigación.

1.3 Hardware and Software.

En el caso del hardware, es requerida una computadora personal simple con las características mínimas para el procesamiento y guardado de información y para le ejecución de los distintos enfoques y algoritmos desarrollados durante la investigación, por esta razón se usó una laptop con las siguientes características:

- Dell XPS M1210
- Core 2 Duo Intel 2Ghz
- 3 GB of RAM

Por el lado del software, varias herramientas fueron ocupadas para lograr distintas tareas. En las siguientes secciones detallaremos cada una de estas herramientas.

1.3.1 CECIL.

CECIL es una librería escrita por Jb Evain [13] con el objetivo de generar e inspeccionar programas y librerías que siguen el formato ECMA CIL [14]. En la última versión, tiene soporte completo para tipos genéricos y los formatos de símbolos para depuración de código. En otras palabras, con CECIL tenemos la capacidad de cargar

ensamblados administrados, navegar a través de sus tipos, modificarlos al vuelo, y volver a guardar el ensamblado modificado en el disco duro.

El API proveído por Afterex depende totalmente de CECIL, ya que este tiene todos los métodos necesarios para modificar e inyectar código CIL.

Otra opción posible en lugar de CECIL, hubiera sido trabajar con el espacio de nombres, System.Reflection y System.Reflection.Emit de la librería de clases base del .NET Framework. Estos espacios de nombre contienen clases que tienen la capacidad de recuperar la información acerca de los ensamblados, como sus módulos, miembros, parámetros y otras entidades, haciendo uso de los metadatos que contienen [11]. Incluso con System.Reflection.Emit se pueden crear tipos dinámicamente. Sin embargo, aunque estos espacios de nombres son una excelente opción para sus propósitos principales, como lo son implementar arquitecturas basadas en plug-ins, inyección de dependencias, y otros patrones dinámicos [12], estos no son la mejor opción para el análisis de código e inyección de código estático.

Reflection no puede considerar código como simples datos, lo cual resulta en varias limitaciones como [10]:

- No se puede descargar un ensamblado una vez que este ha sido cargado en un AppDomain.
- Tiene un rendimiento pobre.
- Consume demasiada memoria.

- No se puede cargar 2 versiones diferentes de un mismo ensamblado en un AppDomain.
- No se puede cargar un ensamblado de Microsoft que tenga una versión diferente a la versión del CLR que se esté ocupando.

Además, System.Reflection sufre de otras limitaciones de diseño, como [10]:

- No analiza código CIL.
- No hace la distinción entre TypeRef y TypeDef.

Por estas razones, pensamos que CECIL es una librería más adecuada para nuestros objetivos, aunque System.Reflection fue muy importante para la implementación de otras partes del sistema, como el patrón inyección de dependencias así como la arquitectura para plug-ins.

1.3.2 Visual Studio Team System 2008.

Visual Studio Team System 2008 es un Ambiente de Desarrollo Integrado (IDE), que provee a miembros de un equipo multidisciplinario un conjunto de herramientas integradas para la arquitectura, diseño, desarrollo, y pruebas de aplicaciones [16]. Todas estas herramientas permiten a los miembros de un equipo, colaborar continuamente y utilizar el conjunto de herramientas como guías en cada paso del ciclo de vida de la aplicación.

Visual Studio fue el IDE primario para todas las tareas de desarrollo en esta investigación. También fue muy importante para la sincronización con los servidores TFS, que administran el código fuente de las distintas versiones del programa.

1.3.3 TFS (Team Foundation Server).

TFS es la oferta de Microsoft para el control del código fuente, colección de datos, reportes y avances de un proyecto en ambientes colaborativos de desarrollo de software. Está disponible como software autónomo o como plataforma cliente servidor para VSTS [19].

TFS fue el sistema de versionado para el desarrollo de Afterex. Los servidores fueron proporcionados por el proyecto codeplex.com.

1.3.4 .NET Framework 3.5.

La versión escogida del .NET Framework fue la 3.5 que fue liberada el 19 de Noviembre del 2007 [18]. Así como con el .NET Framework 3.0, la versión 3.5 usa la versión 2.0 del CLR. Además, instala el .NET Framework 2.0 SP y el .NET Framework 3.0 SP1, los cuales añaden más métodos y propiedades a las clases del BCL en la versión 2.0, que son requeridas para algunas características del 3.5 como LINQ. Estos cambios no afectan a las aplicaciones escritas para el 2.0.

El código fuente del BCL (Base Class Library) en esta versión fue parcialmente liberado para referencia de depuración bajo la licencia MRSL (Microsoft Reference Souce License) [20].

La razón más importante de porque escogimos el .NET Framework 3.5, además de ser la versión más nueva, fue porque alguna de las características introducidas con este framework como expresiones lambda, LINQ para XML, y extensiones de métodos fueron usados ampliamente para crear una aplicación consistente.

1.3.5 Reflector.

Reflector es una herramienta que permite ver, navegar y buscar a través de la jerarquía de clases de un ensamblado de una manera fácil e intuitiva, incluso si no se tiene el código fuente de él. Con reflector se puede de-compilar y analizar ensamblados .NET escritos en C#, Visual Basic y CIL.

Los usos comunes de Reflector son:

- Explorar ensamblados .NET de una manera fácil y natural de entender.
- Entender las relaciones entre clases y métodos.
- Encontrar dónde los tipos son instanciados y expuestos.
- Checar que tu código ha sido correctamente ofuscado antes del lanzamiento.

Además, reflector tiene más de 30 aplicaciones que extienden su funcionamiento, y que pueden ser usados libremente encima de Reflector [21].

Reflector es una aplicación muy útil porque provee un árbol que puede servir para detallar el código CIL de un método y ver que otros métodos llaman al método. También es útil para encontrar las dependencias de ensamblados. Finalmente sirvió para verificar que el código emitido durante la refactorización era correcto.

1.3.6 Reflexil.

Reflexil es un editor de ensamblados que corre como plug-in de Reflector. Reflexil permite manipular código CIL y salvar las modificaciones de vuelta en el disco duro. Reflexil también soporta inyección de código al vuelo de C# y Visual Basic .NET [22].

Una de las virtudes de Reflexil es que permite las modificaciones de ensamblados usando CECIL. Por lo cual sirvió de código ejemplo de cómo hacer uso de esta librería.

1.3.7 Ildasm.

Ildasm ó desensamblador MSIL es una herramienta que acompaña al ensamblador MSIL. Ildasm.exe toma un archivo PE que contiene código CIL y crea un archivo de texto adecuado para ser entrada de ilasm.exe [23]. Ildasm solo trabaja con archivos PE en disco duro, no opera con archivos instalado en el GAC.

El archivo producido por Ildasm.exe es usado como entrada para el ensamblador MSIL. Por otro lado, se puede usar la interfaz gráfica por defecto de Ildasm para ver los metadatos y el código desensamblado de cualquier archivo PE en un árbol jerárquico.

1.3.8 Framework Design Studio.

Framework Design Studio es un conjunto de herramientas para desarrolladores de librerías reusables. El paquete contiene una herramienta gráfica para ver, revisar y comparar diferentes versiones de API´s administrados. También contiene una herramienta de línea de comando para la generación de reportes de diferencias [24].

Esta herramienta nos ayudó para asegurarnos que cuando realizamos una refactorización no rompiéramos ningún método del API original de la aplicación.

1.3.9 PEVerify.

La herramienta PEVerify ayuda a los desarrolladores que generan código CIL para determinar si el código generado así como los metadatos cumplen los requerimientos básicos, como por ejemplo que sean tipos seguros. Algunos compiladores generan tipos seguros solo si se evita usar ciertos aspectos del lenguaje de programación. Si como

desarrollador, tu estas usando un compilador así, y necesitaras verificar si tu código no ha comprometido la seguridad de sus tipos, con PEVerify puedes hacerlo [25].